Formal Foundation for Pattern-Based Modelling

Paolo Bottoni[1], Esther Guerra[2], and Juan de Lara[3]

[1] Università di Roma "Sapienza" (Italy), bottoni@di.uniroma1.it
[2] Universidad Carlos III de Madrid (Spain), eguerra@inf.uc3m.es
[3] Universidad Autónoma de Madrid (Spain), jdelara@uam.es

Abstract. We present a new visual and formal approach to the specification of patterns, supporting pattern analysis and pattern-based model completion. The approach is based on graphs, morphisms and operations from category theory and exploits triple graphs to annotate model elements with pattern roles. Novel in our proposal is the possibility of describing (nested) variable submodels, as well as inter-pattern synchronization across several diagrams (e.g. class and sequence diagrams for UML design patterns). We illustrate the approach on UML design patterns, and discuss its generality and applicability on different types of patterns, e.g. workflow patterns using Coloured Petri nets.

1 Introduction

Patterns are increasingly used in the definition of software frameworks, as well as in Model Driven Development, to indicate parts of required architectures, drive code refactorings, or build model-to-model transformations. The full realisation of their power is however hindered by the lack of a standard formalization of the notion of pattern. Presentations of patterns are typically given through natural language, to explain their motivation, context and consequences; programming code, to show usages of the pattern; and diagrams, to communicate their structure and behavior. However, the use of domain modelling languages, such as UML for design patterns, or Coloured Petri Nets for workflows, forces pattern proposers to provide only examples of their realisation, appealing to intuition to extend them to the complete semantics of the pattern. For instance, the fact that in the *Visitor* pattern there must be a distinct operation in the `Visitor` interface for each `ConcreteElement` is only understood through generalisation of the examples or reading the associated text [4].

The search for a general definition of patterns, independent from the specific modelling language, has led to several proposals based on the association of constraints to diagrammatic definitions, or to extensions of the UML meta-model, in order to distinguish roles from concrete modelling types. However, these proposals incur some problems to define the relations between different components of a patterns specification, as will be illustrated in Section 2.

We propose a formal notion of pattern, grounded in category theory [9], which sees them as formed of: i) a vocabulary of roles; ii) a collection of diagrams, possibly in different modelling languages, defining cardinalities and associations between roles, and supplemented with indications of variable regions; iii)

a collection of interfaces between these diagrams, to identify roles across different diagrams. Two main results are thus obtained, representing a distinguishing novel feature of the approach. First, dependencies among different components of a pattern specification can be formally defined. Second, a clear specification of the parts of the pattern which can be replicated (and of the admissible number of replicas) is given, without recurring to concrete examples. As our notion of pattern generalises patterns from different fields, it opens the way to the extension of pattern-based techniques to new areas, to the formalisation of existing ones, and in general, to the development of pattern-based languages.

The paper is organized as follows. Section 2 presents related approaches. Section 3 introduces our new definition of pattern. Section 4 shows the procedure for pattern application. Finally, Section 5 ends with the conclusions.

2 Related Work

The shortcomings of presenting patterns in the GoF (Gang of Four [4]) style have been addressed by several researchers, who advocate a more formal approach. For example, [3] extends the UML meta-model for class diagrams with specific roles and constraints. Conformance of a model to a pattern is checked as the usual model/meta-model relationship. The technique works for UML class and sequence diagrams, and the emphasis is on specification of patterns, but not on their use for model completion. A non-uniform interpretation of interaction diagrams is provided, where reference to interaction fragments is translated to their unfolding with respect to the instantiation of roles.

The limitations posed by reference to UML diagrams, in particular as regards premature commitment to hierarchical structures, are overcome in [10] by extending the UML meta-model with stereotypes accounting for possible realizations of a given pattern. A distinction between roles, types and classes is used in [8] to decouple representations of roles, at which level the semantics of the pattern is abstractly described by incorporating constraint diagrams into the UML notation, from their refinement as types, and their implementation into classes. This way, the GoF presentation of patterns is shown to be a realization of the abstract level. However, commitment to names and multiplicities is already needed at the type level, with the class level providing concrete implementations. Our proposal, on the contrary, provides the separation through triple graphs which establish a correspondence between roles in a pattern model and types in a pattern specification. Variability is thus maintained also at the type level, leaving matching morphisms to provide the relation with any given realization of the pattern. Independence from UML is achieved in [7] by using Object-Z, but it is limited to structural aspects. Constraints on the use of patterns are exploited in [14] to maintain the consistency of a pattern-based software framework through the use of high-level transformations specific to each pattern.

In [1] a method is proposed for visualizing the roles of the elements in UML class and communication diagrams. The technique extends the UML Profile meta-model with stereotypes and tagged values for pattern annotation. This

also allows pattern composition through single instances. In [6] the intent of design patterns is described with an ontology, which can be queried to obtain suggestions about the most appropriate pattern solving a certain problem.

In [13], the authors propose a logic-based approach, using subsets of First Order Logic – for structural aspects – and Temporal Logic of Actions – for behavioral ones – and supporting pattern combinations. As the language does not include implications, support for complex constraints appears limited.

Graph transformation [2] has also been used to formalise patterns. In [11], patterns are represented with rules, applied to abstract syntax trees to annotate the pattern instances found. In [12], models are transformed to conform to patterns, after having exploited graph queries that detect needs for transformations. In [16] Spatial Graph Grammars provide a graph representation of GoF patterns, to transform object structure graphs so that they conform to patterns. Although declarative, the rules are based on a concrete presentation of patterns, and not on a meta-model characterisation.

In general, we observe the lack of an integrated, domain-independent formalism able to give account of mutual synchronization constraints within a pattern and across different ones, and to support pattern checking, identification and application. In the rest of the paper we present such a formalism.

3 Pattern Specification

3.1 Variable Patterns

We define a variable pattern as made of a fixed part *root* and a number of variable parts V_i, which can be replicated according to a given interval $(low, high)$. Variable patterns support nesting – variable parts inside variable parts – and can be used with any graph model, from unattributed graphs $G = (V_G, E_G, src, tgt : E_G \to V_G)$, to typed node and edge attributed graphs (e.g. E-graphs [2]). Here we use typed attributed graphs and injective morphisms.

Definition 1 (Variable Pattern). *A variable pattern is defined as* $VP = (P = \{V_i\}_{i \in I}, root \in P, int\colon P \to \mathbb{N} \times (\mathbb{N} \cup \{*\}), Emb = \{v_{i,j}\colon V_i \to V_j | \exists V_i, V_j \in P\})$*, where* V_i *are non-empty graphs,* $v_{i,j}$ *are injective morphisms, Emb is a tree with graphs* $V_i \in P$ *as nodes and morphisms* $v_{i,j}$ *as edges, rooted in* $root \in P$*, and int is a function returning the variability interval for each graph* $V_i \in P$*.*

A finite set S satisfies interval (l, h), written $S \lhd (l, h)$, iff $|S| \geq l$ and $h = * \vee |S| \leq h$. Note that if (l, h) is such that $l > h$, then it cannot be satisfied. **Example**. The GoF *Observer* pattern captures one-to-many dependencies between objects so that when the *subject* object changes its state, all the dependent *observer* objects are notified and updated automatically. Fig. 1 presents a simplified version of this pattern using Def. 1. The left part directly encodes the theoretical form, with full definition $VP = (P = \{V_{Ob}, V_{Conc}\}, root = V_{Ob}, int = \{(V_{Ob}, (0, *)), (V_{Conc}, (1, *))\}, Emb = \{v_{Ob,Conc}\})$. The variability interval is related to the satisfiability of the pattern by a model. V_{Ob} has $(0, *)$ as variability

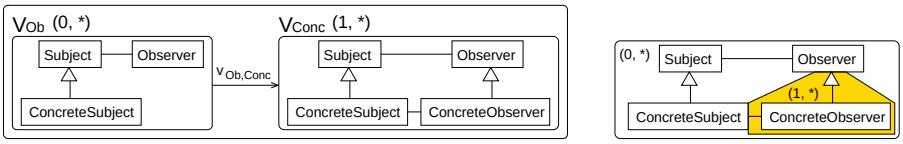


Fig. 1. The Observer Pattern in Theoretical (left) and Compact (right) Forms.

interval, thus the pattern is not mandatory and we may have any number of instances. As $int(V_{Conc}) = (1, *)$, we require at least one `ConcreteObserver` in each pattern instance. The right part of the figure shows a compact form, where fixed and variable parts are presented together. In the figure, we use the concrete syntax of UML class diagrams, but the abstract syntax can also be used. ■

The next definition states when a given graph satisfies a variable pattern.

Definition 2 (Pattern Satisfaction). *Given a graph G and a variable pattern VP as in Def. 1, G satisfies VP, written $G \models VP$, iff:*

- $M_{root} = \{p^k_{root} : root \to G\} \lhd int(root)$.
- $\forall v_{i,j} : V_i \to V_j \in Emb$:
  - $\forall p^k_i \in M_i,\ M^k_j = \{p^l_j : V_j \to G | p^k_i = p^l_j \circ v_{i,j}\} \lhd int(V_j)$.
  - *Define* $M_j = \bigcup M^k_j$, *with* $k = 1..|M_i|$.

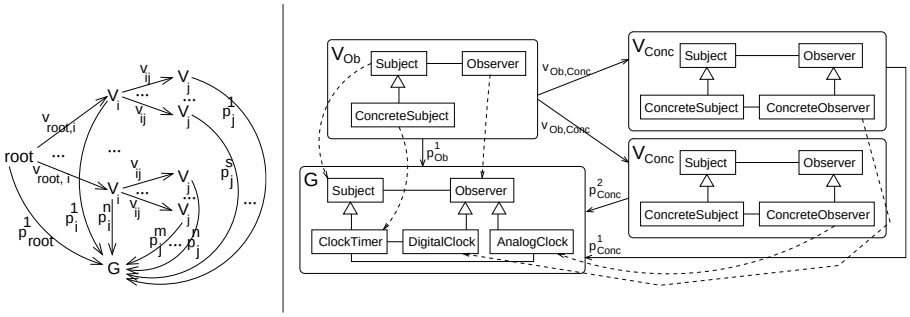


Fig. 2. Pattern Satisfaction (left). Pattern Satisfaction Example (right).

Remark. The procedure in Def. 2 induces a tree traversal for Emb, because when $V_i \to V_j$ is traversed, M_i must have been previously calculated. ■

The left of Fig. 2 describes the morphisms for the satisfaction checking for a variable pattern with one level of nesting, i.e. $Emb = \{root \to V_i, V_i \to V_j\}$. We assume one morphism $p^1_{root} \in M_{root}$. The procedure checks that the set M^1_i of all morphisms $p^1_i, ..., p^n_i : V_i \to G$ commuting with $G \overset{p^1_{root}}{\longleftarrow} root \overset{v_{root,i}}{\longrightarrow} V_i$ satisfies the interval $int(V_i) = (low_i, high_i)$. Similarly, given each morphism $p^k_i \in M_i = M^1_i$,

each set M_j^k of morphisms $p_j^1, ..., p_j^s \colon V_j \to G$ commuting with $G \overset{p_i^k}{\leftarrow} V_i \overset{v_{i,j}}{\rightarrow} V_j$ satisfies the interval $int(V_j)$. In the figure, we have replicated V_j and V_i for each different morphism p_j^k and p_i^k, to show more intuitively the tree traversal.
Example. The right of Fig. 2 shows a model G that satisfies the specification of the observer pattern of Fig. 1. The fixed part V_{Ob} occurs once, and the variable region V_{Conc} twice. This is checked by building the set $M_{Conc} = \{p_{Conc}^1 \colon V_{Conc} \to G, p_{Conc}^2 \colon V_{Conc} \to G\} \lhd (1, *)$. In the figure, the dotted lines indicate some of the mappings induced by p_{Ob}^1, p_{Conc}^1 and p_{Conc}^2. ■

3.2 Annotating Structure with Roles: Triple Patterns

In order to specify a pattern, we need its structure (as given by a variable pattern), a vocabulary of pattern roles, and a mapping from the elements in the pattern to the vocabulary. We call this structure *annotated pattern* and ground it in the notion of triple graph [5], composed of two graphs, *source* and *target*, related through a *correspondence* graph. Nodes in the correspondence graph (also called *mappings*) have morphisms to nodes or edges in the other two graphs. Thus, mappings may relate two nodes, two edges, an edge and a node, or just point to a single element in the source or target graphs [5].

Definition 3 (Triple Graph). *A triple graph $TrG = (G_s, G_c, G_t, c_s, c_t)$ has three graphs G_i ($i \in \{s, c, t\}$), and two functions $c_j \colon V_{G_c} \to V_{G_j} \cup E_{G_j} \cup \{\cdot\}$ ($j = s, t$).*

The element "$\cdot$" in the codomain of c_j denotes that the correspondence function c_j can be undefined. A triple graph is also written as $(G_s \overset{c_s}{\longleftarrow} G_c \overset{c_t}{\longrightarrow} G_t)$. We say that a node or edge x of G_s is related to a node or edge y of G_t iff $\exists n \in V_{G_c}$ s.t. $x \overset{c_s}{\longmapsto} n \overset{c_t}{\longmapsto} y$ and we write $x \; rel_{TrG} \; y$. As [5] showed, triple graphs and morphisms form the category[4] **TrGraph** [9].
Example. Fig. 3 shows a triple graph. Its source graph G_s at the bottom conforms to the UML meta-model, its target G_t at the top is a pattern vocabulary model, and the correspondence graph G_c maps UML elements to vocabulary elements. The morphisms from the nodes in G_c are shown as dotted arrows. The correspondence function c_s is undefined for node `:PatternInstance` (i.e. $c_s($`:PatternInstance`$) = \cdot$) and this is represented by omitting the edge. ■

Triple graphs are typed by meta-model triples [5] made of two meta-models, source and target, related through a correspondence meta-model. In our case, a meta-model triple relates the meta-model of a specific language (e.g. UML) with that of a generic pattern vocabulary, which can be refined by subclassification in order to define patterns for the language. The top of Fig. 4 shows the meta-model for the vocabulary. All classes except the subclasses of `PatternRole` allow defining patterns in any language. A `Pattern` has a name and a type, contains participating roles, and can be documented with its motivation, applicability, intent

[4] A category is made of objects (e.g, triple graphs) and arrows (e.g., triple morphisms) satisfying some conditions [9].

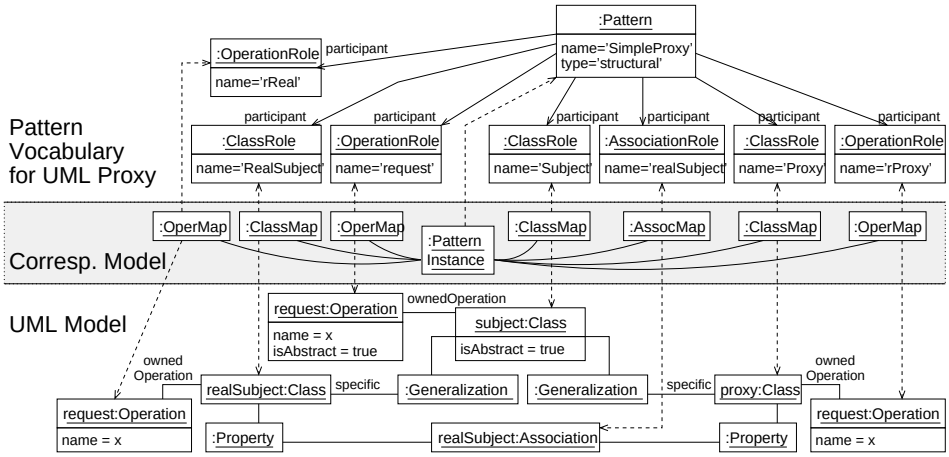


Fig. 3. Annotated Pattern with the Structural Part of the Proxy (in Abstract Syntax).

and consequences. Relations between patterns are given through the `Relation` association class. We have specialized the meta-model for UML, so that roles are applicable to operations, classifiers, classes, structural features and associations. The bottom of Fig. 4 partially shows the UML meta-model, and the correspondence meta-model maps roles to UML elements. The `PatternInstance` class is used to group the mappings of each pattern instantiation.

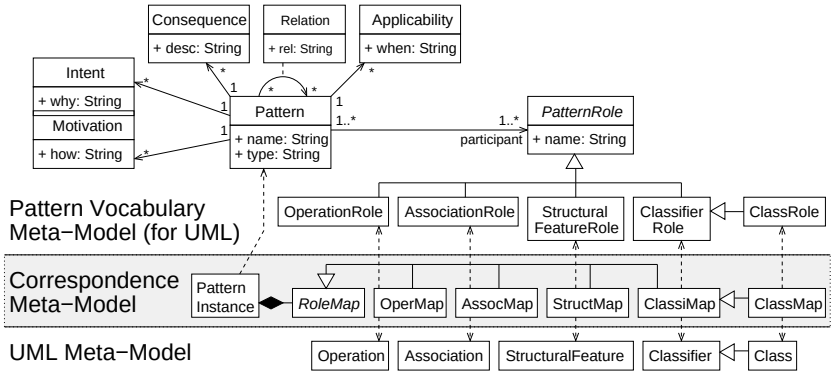


Fig. 4. Meta-model Triple for UML Patterns.

A *pattern-annotated model* is a triple graph whose source is a model in some language (e.g. UML), and the target contains a subgraph isomorphic to a pattern vocabulary, for each pattern used in the source model. The correspondence graph is called *annotation graph* and has a `PatternInstance` node for each pattern instance and one `RoleMap` for each element playing a role in the instance. For

example, the annotated pattern in Fig. 3, specifying the structure of the *Proxy* pattern [4], conforms to the meta-model triple in Fig. 4. The intent of this design pattern is to provide a surrogate or placeholder *proxy* for an object with role *realSubject* in order to control access to it. The bottom part of Fig. 3 uses the abstract syntax for UML class diagrams. For simplicity, we only show the pattern and the roles in the vocabulary model. The pattern has only one fixed graph, no variable part, and requires that the operations in `Subject`, `RealSubject` and `Proxy` have the same name, which is modelled with a variable `x`. More complex attribute conditions such as in [2] are possible. We have omitted the name of classes and associations, so they can be mapped to any name.

An *annotated pattern* is a variable pattern where all the graphs in Def. 1 are triple graphs, and morphisms are triple morphisms. The notion of pattern satisfaction remains as in Def. 2, but using triple graphs. Thus, annotated patterns are satisfied by triple graphs, called pattern-annotated models.

Example. Workflow patterns [15] collect recurring constructs from existing workflow systems and provide descriptions of their usage. Their presentation is textual in the style of GoF patterns. For control flow patterns, dealing with synchronization policies, an intuitive semantics is given through Coloured Petri Nets showing example realizations of the pattern, to be inferred by the reader. Fig. 5 shows three annotated patterns expressed via a syntax identifying the roles of places and transitions, as *input*, *split*, *output*, *and*, and *merging*. ■

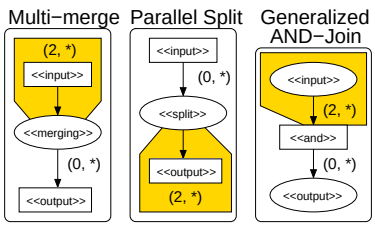


Fig. 5. Workflow Patterns.

As seen before, a tool need not show the triple graph to the user, but the annotation can be done by marking the source graph [1]. However for the theory, an explicit triple graph has some advantages: (i) we do not modify or extend the source meta-model (e.g. the UML one) with additional classes or attributes for tagging; (ii) triple graphs help in enforcing the patterns, as shown in Section 4; (iii) it is easier to distinguish the instances of a pattern, as these are identified by `PatternInstance` nodes.

3.3 Synchronizing Different Variable Patterns

A pattern specification can be composed of more than one diagram. For instance, the GoF patterns are described using class and sequence diagrams. Thus, relationships have to be established between the elements in the different diagrams, which we do through their roles in the pattern.

Example. The *Visitor* pattern explicitly represents as objects the operations to be performed on the elements of an object structure, so that new operations can be defined without changing the classes of the elements on which they operate. It is one of the most complex GoF patterns as it requires two levels of variation for the two hierarchies of `Visitor` (i.e. the operations) and `Element` (i.e. the object structure), with the set of operations for `Visitor` varying together with

the `ConcreteElement` set. This covariance cannot be expressed through a constraint on cardinalities, as it requires an exact match on types of parameters. The pattern is shown in Fig. 6, where the presence in the same variance region of the signatures for the `visit` operations constrains the types of their parameters to satisfy the constraint of equality with the types of `ConcreteElement`.

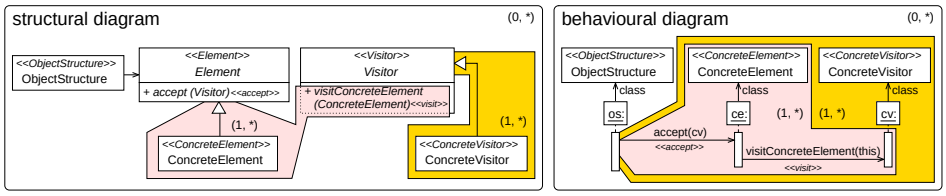


Fig. 6. Visitor Pattern.

Both the structural and the behavioral part of *Visitor* require two variable parts, relative to the two hierarchies. However, while in the structural part the two regions are independent, in the behavioral part they are nested. This reflects the double constraint that each concrete visitor can be accepted by each concrete element and that each concrete visitor has an operation to visit each concrete element. The synchronization between the different regions involved is represented by the equality of the colours used in the different pattern graphs, and formally as a *synchronization graph*.

Fig. 7 shows the scheme of the two patterns for *Visitor*: $SP : V_1^{SP} \leftarrow V_0^{SP} \rightarrow V_2^{SP}$ and $IP : V_0^{IP} \rightarrow V_1^{IP} \rightarrow V_2^{IP}$. SP has V_0^{SP} as root and two variable parts, and describes the structural part (a class diagram). IP has root V_0^{IP} and a variable part V_1^{IP} with nested V_2^{IP}, modelling a sequence diagram. The *synchronization graph* $I_{11} \leftarrow I \rightarrow I_{22}$ declares the intersections between the roots and pairs of variable parts of each pattern. Morphism $I \rightarrow I_{11}$ is derived as we have $V_0^{SP} \rightarrow V_1^{SP}$ and $V_0^{IP} \rightarrow V_1^{IP}$, and similar for $I \rightarrow I_{22}$. All squares in the diagram must commute to ensure coherence, so $I \rightarrow V_0^{SP} \rightarrow V_1^{SP} = I \rightarrow I_{11} \rightarrow V_1^{SP}$, and similar for V_0^{IP} with I_{11} and I_{22}. ■

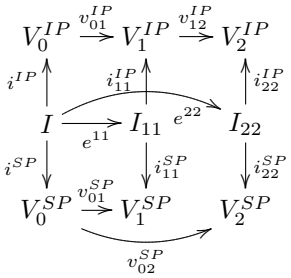


Fig. 7. Synchronization

Example. Fig. 8 shows a variation of the *Proxy* pattern, in concrete syntax, allowing several proxies for a given subject ($V_0^{SP} \rightarrow V_1^{SP}$). The upper part ($V_0^{IP} \rightarrow V_1^{IP}$) shows the annotated sequence diagram. For clarity, we show the classifiers of the `p` and `rs` objects, as both classifiers play a role in the pattern. ■

In Fig. 8 and others, we use a shortcut notation, shown below, for sequence diagrams. The `OperMap` node in the annotation graph points to the message arrow for the invocation, while in the abstract syntax the morphism reaches the operation to be executed through a `MessageOccurrenceSpecification` object.

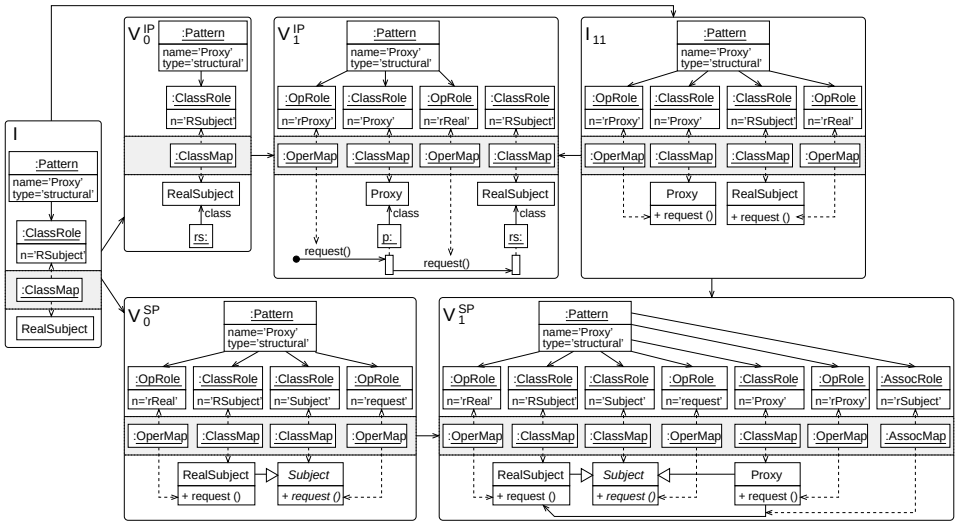
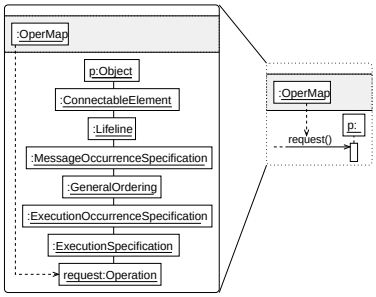


Fig. 8. Synchronization Example for the Proxy Pattern.

The two variable patterns to be synchronized share the same vocabulary model, as they describe different diagrams of the *same* pattern. Given the parts to be synchronized, the synchronization graph is automatically calculated by the intersections w.r.t. the roles. Thus, two elements in two patterns SP and IP, if mapped to the same role, will be present in I.



Once the intersection graphs I_{ij} (and the morphisms $V_i^{SP} \leftarrow I_{ij} \rightarrow V_j^{IP}$) are derived, the morphisms of the synchronized graph are derived as well. Morphism $I_{ik} \rightarrow I_{jl}$ is added, iff there is a path $V_i^{SP} \rightarrow V_j^{SP}$ in the Emb tree of SP and a path $V_k^{IP} \rightarrow V_l^{IP}$ in the Emb tree of IP.

Definition 4 (Synchronization Graph). *Let* $AP^k = (P^k = \{V_i^k\}_{i \in I^k}, root^k \in P^k, int^k, Emb^k = \{v_{i,j}^k \colon V_i^k \rightarrow V_j^k\})$, $k = \{1, 2\}$, *be two annotated patterns on the same vocabulary* VP*; their* synchronization graph $SG = (V = \{root^1 \overset{i^1}{\leftarrow} I \overset{i^2}{\rightarrow} root^2, ..., V_i^1 \overset{i_{ij}^1}{\leftarrow} I_{ij} \overset{i_{ij}^2}{\rightarrow} V_j^2, ...\}, E = \{e_{ij}^{kl} \colon I_{ij} \rightarrow I_{kl}\})$ *is calculated as follows:*

- *For each pair of triple graphs* $V^k = (V_s^k \leftarrow V_c^k \rightarrow V_t^k)$ *to be synchronized:*
  - *Build the triple graph* $O = (O_s \leftarrow O_c \rightarrow VP)$*, and triple morphisms* $V^1 \xrightarrow{f^1} O \xleftarrow{f^2} V^2$*, jointly surjective on* O_s *and* O_c *as follows: (i)* VP *is the common vocabulary model, so there are inclusions* $V_t^k \hookrightarrow VP$ *(ii) if two elements* $a \in V_s^1$, $b \in V_s^2$ *are mapped to the same role in* VP*, (i.e.* a rel_{V^1} x, b rel_{V^2} x*) only one element* ab *is added to* O_s*, and the functions*

f^1 and f^2 identify a and b to such element, $f^1(a) = ab = f^2(b)$ (if ab does not exist, the graphs cannot be synchronized).

- *Intersection I is given by the pullback* [5] *$V^1 \xleftarrow{i^1} I \xrightarrow{i^2} V^2$ of $V^1 \xrightarrow{f^1} O \xleftarrow{f^2} V^2$.*

– *Given two intersection nodes $V^1_r \leftarrow I_{rs} \rightarrow V^2_s$ and $V^1_u \leftarrow I_{uv} \rightarrow V^2_v$ s.t. there are paths $V^1_u \rightarrow V^1_r$ in Emb^1 and $V^2_v \rightarrow V^2_s$ in Emb^2, add edge $e^{rs}_{uv}\colon I_{uv} \rightarrow I_{rs}$ to SG, as morphism e^{rs}_{uv} uniquely exists due to the pullback universal property [2], see the left of Fig. 9.*

Remark. Two graphs V^1 and V^2 can be synchronized only if their variability intervals overlap (i.e. $\exists M | M \lhd int^1(V^1)$ and $M \lhd int^2(V^2)$).

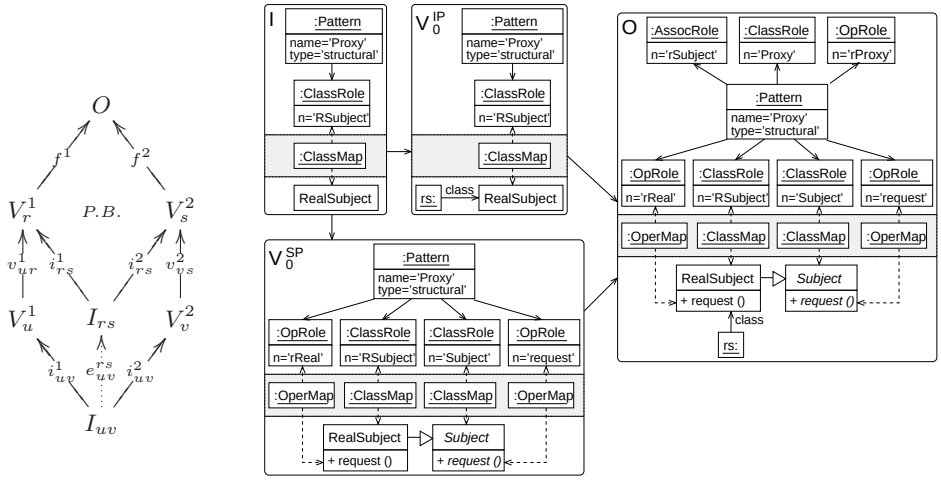


Fig. 9. Calculating Intersection Edges (left). Example (right).

Example. The right of Fig. 9 shows an example of the calculation of an intersection graph for the synchronization scheme shown in Fig. 8. ■

3.4 Full Pattern Specification

We now put all elements together to define a full pattern, made of a primary *structuring* pattern SP, and a number of secondary patterns IP_i synchronized with SP through synchronization graphs SG_i.

Definition 5 (Full Pattern Specification). *A* Full Pattern Specification $PS = (VP, SP, Sec = \{(IP_s, SG_s)\}_{s \in S})$ *is composed of:*

[5] Roughly, a pullback [9] is the biggest intersection of two objects A_i through a common one B to which both are mapped. The pullback $A^1 \leftarrow C \rightarrow A^2$ identifies the elements of A^1 and A^2 that are mapped to a common element in B.

1. *A Pattern Vocabulary VP, mentioning the relevant roles for the patterns.*
2. *A Structuring Pattern Diagram $SP = (P^{SP} = \{V_i^{SP}\}_{i\in I}, root^{SP}, int^{SP}, Emb^{SP} = \{v_{i,j}^{SP} : V_i^{SP} \to V_j^{SP}\})$, an annotated pattern where the V_i^{SP} are triple graphs whose target is a subgraph of the pattern vocabulary VP.*
3. *A set Sec of pairs made of a Secondary Pattern Diagram $IP_s = (P_s^{IP} = \{V_i^{IP}\}_{i\in I_s}, root_s^{IP}, int_s^{IP}, Emb_s^{IP} = \{v_{i,j}^{IP} : V_i^{IP} \to V_j^{IP}\})$ and a Synchronization Graph $SG_s = (V_s = \{V_i^{SP} \overset{i_{ij}^{sp}}{\leftarrow} I_{ij} \overset{i_{ij}^{ip}}{\rightarrow} V_j^{IP}\}, E_s = \{e_{ij}^{kl} : I_{ij} \to I_{kl}\})$, which synchronizes the secondary patterns with the primary one.*

For UML patterns, the primary pattern SP is a class diagram, and Sec generally contains a sequence diagram synchronized with the class diagram.

The satisfaction of full patterns is similar to Def. 2, but using annotated patterns and taking into account the synchronization graphs.

Definition 6 (Full Pattern Satisfaction). *Given a pattern-annotated model $TrG = (G_s \overset{c_s^G}{\longleftarrow} G_c \overset{c_t^G}{\longrightarrow} G_t)$, and a full pattern specification $PS = (VP, SP, \{(IP_s, SG_s)\}_{s\in S})$ as in Def. 5, TrG satisfies PS, written $TrG \models PS$, iff:*

- *$\forall (IP, SG) \in Sec$ (i.e. for each secondary pattern and synchronization graph):*
  - *Let $M_{root,root} = \{root^{SP} \overset{ps^k}{\rightarrow} TrG \overset{pi^k}{\leftarrow} root^{IP} | root^{SP} \overset{i^{sp}}{\leftarrow} I \overset{i^{ip}}{\rightarrow} root^{IP}$ is pullback$\}$, then $M_{root,root} \lhd int^{SP}(root^{SP})$ and $M_{root,root} \lhd int^{IP}(root^{IP})$.*
  - *$\forall e_{jk}^{lm} : I_{jk} \to I_{lm} \in E$ (i.e. for each edge in SG):*
    - *$\forall V_j^{SP} \overset{ps^r}{\rightarrow} TrG \overset{pi^r}{\leftarrow} V_k^{IP} \in M_{j,k}$, let $M_{l,m}^r = \{V_l^{SP} \overset{ps^u}{\rightarrow} TrG \overset{pi^u}{\leftarrow} V_m^{IP} | ps^u \circ (V_j^{SP} \to V_l^{SP}) = ps^r$ and $pi^u \circ (V_k^{IP} \to V_m^{IP}) = pi^r$ and $V_l^{SP} \overset{i^{sp}}{\leftarrow} I_{lm} \overset{i^{ip}}{\rightarrow} V_m^{IP}$ is pullback$\}$. Then $M_{l,m}^r \lhd int^{SP}(V_l^{SP})$ and $M_{l,m}^r \lhd int^{IP}(V_m^{IP})$. See Fig. 10(a).*
    - *Define $M_{l,m} = \bigcup M_{l,m}^r$, with $r = 1..|M_{j,k}|$.*

Example. Fig. 10(b) shows in compact notation the satisfaction of the *Proxy* pattern of Fig. 8. The model contains a class diagram and two sequence diagrams, enclosed in different regions, which a tool would present in three views. The model has one instance of the *Proxy* pattern, and the variability region that affects the `Proxy` role has been instantiated twice (classes `ImageProxy` and `RemoteProxy`). Fig. 10(c) shows the first step in the satisfaction checking, where the pullback of the fixed parts is depicted, corresponding to the intersection graph shown to the right of Fig. 9. The satisfaction check follows by computing two additional pullbacks for the two instantiations of the variable parts. ■

4 Pattern-Based Model Completion

Patterns as defined above can be used in different scenarios: (i) to query how many instances of each pattern a model contains, or to analyse pattern instance interactions; (ii) for pattern extraction; (iii) to check whether a part of the

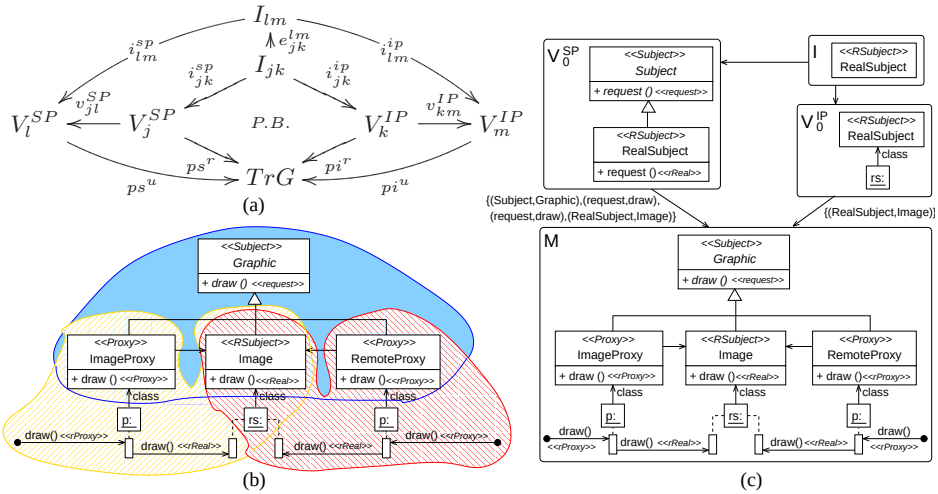


Fig. 10. (a) Satisfaction of Full Pattern, where Outer Square is Pullback. (b) Annotated Model Satisfying the Proxy Pattern. (c) First Step in Satisfaction Checking.

model (maybe created by hand) conforms to a pattern; and (iv) to automatically complete a model according to a pattern. In this section we concentrate on the latter, giving the algorithm for pattern application for model completion and proving its correctness.

We start by showing how to apply the primary pattern, and then present how synchronization with the secondary patterns is achieved. Given a pattern $(VP, SP, Sec = \{(IP_s, SG_s)\}_{s \in S})$, the application of the primary pattern SP to an annotated model $M = (M_s \leftarrow M_c \rightarrow M_t)$ is as follows:

1. **Vocabulary Extension.** Add the vocabulary of the pattern to M_t if it was not added before (i.e. this is the first instance of the pattern).
2. **Role Annotation.** The user selects in M_s the elements playing some role in the pattern. Thus, a `RoleMap` node is created in M_c for each of these elements, associated with a node pm of type `PatternInstance`. pm is a new node for a new instance of the pattern, or an existing one, if extending a previous instance. Construct the morphisms from the modified annotation graph M_c to M_t and M_s. A new instance cannot be created if the number of existing instances would exceed the interval for the pattern root.
3. **Instance Extraction.** Construct a pattern graph PG by navigating from pm to the elements in M_c belonging to the defined instance of the pattern, and from these to elements in M_t and M_s along the c_s and c_t morphisms.
4. **Variability Instantiation.** The user selects a number r_i in the range $int(V_i) = (l_i, h_i)$ for each variable part V_i of the pattern, such that the existing number of instances e_i plus the new ones r_i satisfy the interval.

Build a graph PC as the colimit[6] of the fixed part of the pattern P and $r_i + e_i$ instantiations of each variable part.

5. **Model Extension.** Construct the pushout[7] M' of PC and M through PG.

Example. Fig. 11 shows the application of a pattern with variable part V_1 and nested part V_2. Once V_1 and V_2 are instantiated, we build the colimit PC and the pushout M' of $PC \leftarrow PG \rightarrow M$.

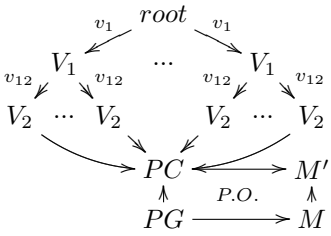


Fig. 11. Application of Structural Pattern.

Fig. 12 shows the application of the Proxy to a model M containing a class `Image`, which the user mapped to role `RealSubject`. The name of the operation in the pattern ("request", a variable) is mapped to the name of the operation in the model ("draw"), and similarly for class names. The user selected two instantiations of the variable part; hence two proxies are created in the resulting model M'. In the pushout, the new elements may contain variables, like the name of the `Proxy` class to be added. In this case, either the user provides a value (`ImageProxy` and `RemoteProxy` in the example), or default ones are obtained from the role names. ■

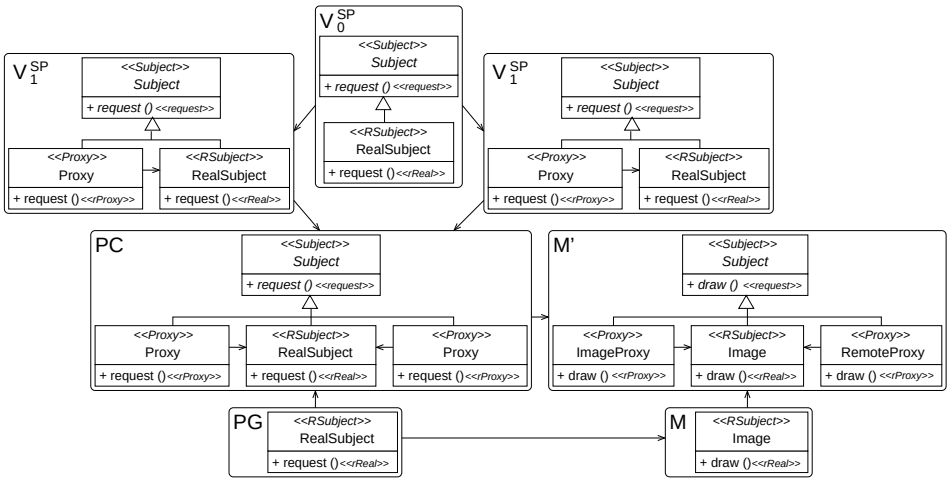


Fig. 12. Applying a Structural Pattern to an Annotated Model.

The application of the secondary patterns continues by enlarging the model M' obtained in the previous procedure by a sequence of pushouts. Hence, we first

[6] Roughly, a colimit [9] is the smallest object in which a diagram made of objects and morphisms is embedded (assuming injective morphisms).

[7] Given two objects A_i, whose "intersection" is given by $A_1 \leftarrow I \rightarrow A_2$, the pushout $A_1 \rightarrow B \leftarrow A_2$ is their union, where the common elements (given by I) are "merged".

check which variable parts V_i^{SP} were added to M to yield M' (steps 2a and 2b). This is necessary as the user may have extended an existing instance. Then, the synchronization graph is used to locate the variable part of the secondary pattern V_l^{IP} synchronized with V_i^{SP}, and a pushout is built. We use an intermediate graph B_{kl}^{IP} (step 1) as pushout object, as it contains the intersection between V_l^{IP} and the previous graph in the nesting structure, preventing the addition of too many elements. The procedure is repeated for each secondary pattern:

1. $\forall e_{jk}^{il}$: $I_{jk} \rightarrow I_{il}$, edge in the synchronization graph, calculate the pushout object B_{kl}^{IP} as shown in Fig. 13(left). Morphism $B_{kl}^{IP} \rightarrow V_l^{IP}$ exists due to the pushout universal property. Note also that if there are morphisms $V_k^{IP} \rightarrow M'$ and $I_{il} \rightarrow M'$, then we uniquely have $B_{kl}^{IP} \rightarrow M'$ due to the pushout universal property. Graph B_{kl}^{IP} will be used later in this procedure.
2. **For each** $(IP, SG) \in Sec$ **do** (i.e. for each sec. pattern and synch. graph):
   - Traverse the graphs used to build the colimit PC in step 4 of the previous procedure (see Fig. 11) in depth first order.
     (a) **Let** V_i^{SP} be the current node; calculate the pullback object X of $V_i^{SP} \rightarrow PC \leftarrow PG$.
     (b) **If** $X \ncong V_i^{SP}$ **then**
        i. **If** $V_i^{SP} = root^{SP}$ **then** update the model M' according to the pushout shown in the center of Fig. 13.
        ii. **Else let** V_j^{SP} be the predecessor of V_i^{SP} in Emb^{SP} and update the model M' according to the diagram to the right of Fig. 13.
        iii. **Repeat**
        - **Let** V_m^{IP} be the child of V_l^{IP} in Emb^{IP}. Update the model (like in steps i and ii) for each instance V_n^{SP} s.t. $V_n^{SP} \leftarrow I_{nm} \rightarrow V_m^{IP}$, and whose pullback object X in $V_n^{SP} \rightarrow PC \leftarrow PG$ is not isomorphic to V_n^{SP}.

        Until all descendants of $V_l^{IP} \in Emb^{IP}$ have been visited.

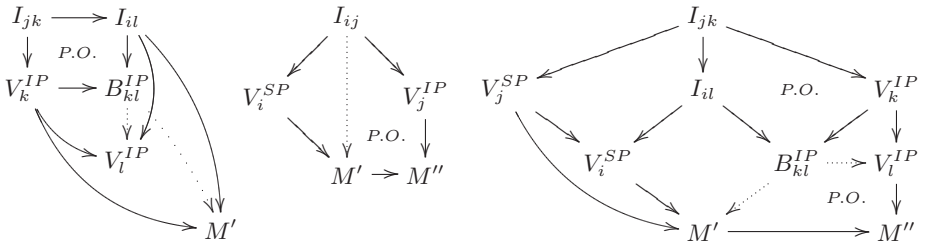


Fig. 13. Building B_{kl}^{IP} (left). First (center) and Second (right) Cases for Model Update.

The procedure is incremental – one can update an existing instance – and supports heterogeneous synchronization, e.g. Fig. 7, where the structural pattern has two independent variable parts and nesting in the interaction pattern.

Example. For *Proxy*, Fig. 14 shows how the first sequence diagram is created starting from M' of Fig. 12. Note that, as there were two instantiations of V^{SP}, the procedure would follow by adding an additional sequence diagram. ■

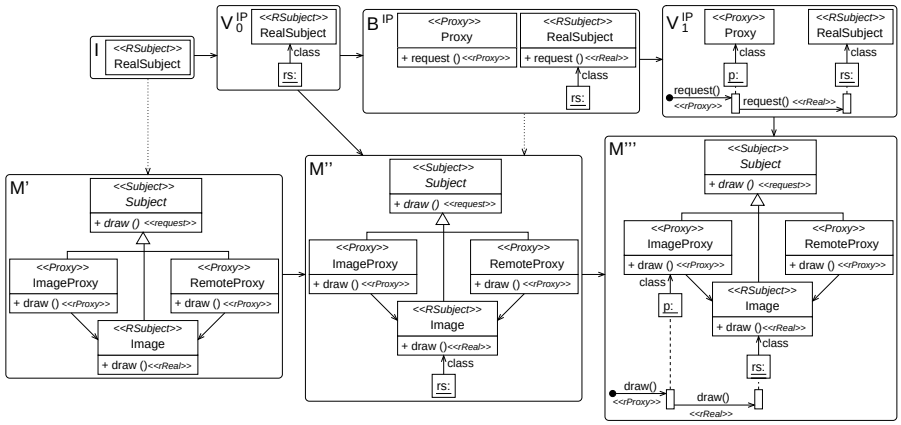


Fig. 14. Synchronization: Building the First Sequence Diagram.

Finally, it can be shown that, after applying a pattern according to the previous procedure, the resulting model satisfies such pattern according to Def. 6.

Proposition 1 (Application Correctness). *If model $M^{k'}$ is obtained from M by applying pattern SP according to the previous procedure, then $M^{k'} \models SP$.*

Proof Sketch. The nodes I_{ij} of SG are the pullbacks of $V_i^{SP} \rightarrow O \leftarrow V_j^{IP}$. When applying the primary pattern, the vocabulary model is added to M, and after the pushouts for the secondary pattern are made, yielding $M^{k'}$, we have $O \rightarrow M^{k'}$. This implies that all I_{ij} are the pullback objects of $V_i^{SP} \rightarrow M^{k'} \leftarrow V_j^{IP}$. The satisfaction $M^{k'} \models SP$ then follows as the application procedure takes care of the correctness of the replications w.r.t. the allowed variability intervals (step 4 of the application of the primary pattern). ■

5 Conclusions and Future Work

We have presented a formal approach to the specification of patterns, as well as procedures for checking whether a model satisfies a pattern and applying a pattern to a model for model completion. In the proposed formalization, patterns are annotated by roles in a vocabulary, support the synchronization of different types of diagrams, and allow the definition of variable parts, possibly nested. The proposal relies on a general meta-model for patterns, not necessarily based on UML.

Our approach presents several benefits w.r.t. existing ones. First, variability regions – with the possibility of nesting – are more flexible than current proposals, which annotate single elements with cardinalities [3] and for which it is more difficult to express that several elements have to vary together. Second, our mechanism for pattern application considers synchronization of several diagrams. Third, we separate the roles from the pattern structure by a triple

graph, without extending existing meta-models. This clean and non-intrusive solution facilitates the manipulation and querying of the vocabulary models, as well as the identification of pattern instances. Our formal treatment facilitates the derivation of rules for refactoring towards patterns.

We plan to investigate pattern conflicts and reason about pattern interaction effects, e.g. using graph constraints or critical pairs [2] in pattern application.

Acknowledgments. Work supported by the Spanish Ministry of Science and Innovation, projects METEORIC (TIN 2008-02081) and MODUWEB (TIN 2006-09678). We thank the referees for their insightful and detailed comments.

References

1. J. Dong, S. Yang, and K. Zhang. Visualizing design patterns in their applications and compositions. *IEEE Trans. Software Eng.*, 33(7):433–453, 2007.
2. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer, 2006.
3. R. B. France, D.-K. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *IEEE Trans. Software Eng.*, 30(3):193–206, 2004.
4. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
5. E. Guerra and J. de Lara. Event-driven grammars: Relating abstract and concrete levels of visual languages. *Software and System Modeling*, 6(3):317–347, 2007.
6. H. Kampffmeyer and S. Zschaler. Finding the pattern you need: The design pattern intent ontology. In *MoDELS*, volume 4735 of *LNCS*, pages 211–225. Springer, 2007.
7. S. K. Kim and D. Carrington. Using integrated metamodeling to define OO design patterns with Object-Z and UML. In *APSEC*, pages 257–264. IEEE Computer Society, 2004.
8. A. Lauder and S. Kent. Precise visual specification of design patterns. In *ECOOP*, volume 1445 of *LNCS*, pages 114–134, 1998.
9. S. Mac Lane. *Categories for the Working Mathematician. 2^{nd} Edition. Graduate Texts in Mathematics Vol 5.* Springer, 1998.
10. J. K.-H. Mak, C. S.-T. Choy, and D. P.-K. Lun. Precise modeling of design patterns in UML. In *ICSE*, pages 252–261. IEEE Computer Society, 2004.
11. J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *ICSE*, pages 338–348. ACM, 2002.
12. A. Radermacher. Support for design patterns through graph transformation tools. In *AGTIVE*, volume 1779 of *LNCS*, pages 111–126. Springer, 1999.
13. T. Taibi and D. C. L. Ngo. Formal specification of design pattern combination using BPSL. *Information and Software Technology*, 45:157–170, 2003.
14. T. Tourwé and T. Mens. High-level transformations to support framework-based software development. In *SET*, volume 72-4 of *ENTCS*, 2003.
15. W. van der Aalst, A. ter Hoefstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Data Bases*, 14(3):5–51, 2003.
16. C. Zhao, J. Kong, J. Dong, and K. Zhang. Pattern-based design evolution using graph transformation. *J. Vis. Lang. Comput.*, 18(4):378–398, 2007.